2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

The missing SystemC and TLM asynchronous features enabling inter-simulation synchronization.

Guillaume Delbergue
GreenSocs -
Bordeaux INP, CNRS IMS, UMR 5218
guillaume.delbergue@greensocs.com

Mark Burton
GreenSocs
mark.burton@greensocs.com

Bertrand Le Gal and Christophe Jego
Bordeaux INP, CNRS IMS, UMR 5218
bertrand.legal@ims-bordeaux.fr
christophe.jego@ims-bordeaux.fr

***Abstract*—While the TLM-2.0 interface has proved to be both robust and widely applicable, there are some missing features within SystemC. First, enabling multi-core simulation by allowing TLM communication to be synchronized between different simulators (between different hosts, host processes or host threads). Second, synchronizing the time for different quantum domains based on the quantum time, rather than the simulation time. This paper introduces new SystemC features to handle properly different kind of synchronizations between SystemC kernels and with others simulators.**

I. Introduction

Two independent mechanisms to do with synchronization are presented in this paper. First, we propose a modification to the SystemC kernel to allow multiple simulations (either SystemC simulations, or other simulators) to synchronize with the kernel. Second, we propose a addition to the TLM-2.0 Quantum-keeper to allow events to be timed based on the quantum time, rather than the kernel time, hence allowing quantum-domains to synchronize correctly within themselves in a standard way.

Both mechanisms have previously been discussed (for instance in [1]), in this paper we present the refinement that has been needed in order to make the modifications suitable for standardization.

II. Synchronizing Simulators

It is becoming increasingly important to be able to divide simulations between multiple host threads or processors, or even multiple hosts. Different approaches have been taken to this task, for instance attempting to divide the SystemC model itself into parallel threads of execution as presented in [2]. For abstractions where models are highly parallel, closer to the inherent parallelism of RTL and real hardware, this approach has some significant advantages. However, the approach has been largely discussed at higher levels of abstraction. Its extremely hard to achieve this without significant changes to both the SystemC kernel and the models running on the kernel. It is these higher levels of abstraction that are appealing to the industry as they struggle with the ever-increasing complexity of software development based on increasingly complicated hardware designs. Often the designer knows where to (or is forced to) parallelize a model, and very typically this parallelization can be made to match the available host cores. It is therefore, in many circumstances, advantageous to allow the designers to divide the model themselves rather than proposing an automated mechanism.

However, today, such a division requires extensive work to allow synchronization between the different parts, and that mechanism must be used for all parts. It is typically not possible to have two sub-systems using different synchronization mechanisms. The biggest issue faced by such a synchronization mechanism is that the SystemC kernel is expected to exit if it runs out of events to process. In the case of multiple kernels, while overall there may be more events to process, an individual kernel may be starved of events. Typically preventing this requires a system of locks (or spinning the kernel un-necessarily). While a locking mechanism is preferable in terms of CPU usage, it must be used universally, which requires standardization. Moreover, the proposed mechanisms support existing models without modifications.

This paper proposes a mechanism that allows different synchronization schemes to be built, rather than proscribing a single implementation. There is (for instance) no requirement that both ends of the mechanism are running SystemC, nor is any attempt made to standardize the means of communication between the simulators.

That can safely be left to individual engineers as it only effects the individual integration. However, we have taken care to ensure that the mechanisms we propose can be used with a TLM-2.0 communication style as our expectation is that TLM-like interfaces are likely to be the boundary across which model division will occur.

The mechanism is suitable for standardization, and is intended to support future research efforts to develop efficient inter-SystemC synchronization mechanisms.

A. Requirements

The primary requirement is to propose a mechanism by which SystemC can be allowed to wait for an external event from another simulator without terminating the ongoing simulation. We term this Asynchronous Wait. In addition, we require that the mechanism can be used within the context of a TLM-like communication, specifically adhering to the notion of a quantum. If we think about a quantum cycle, typically we can have a number cases between two simulators (A and B):

- Simulator A and B could be executing their respective loads, neither of them having reached a quantum boundary. In this case, a TLM-like communication from one to the other can present as a normal event in the second simulator. This case is already handled today with asynchronous notify.
- Simulator A could be ahead of B, perhaps already arriving at a Quantum boundary and waiting for B. There are two cases here:
  - Simulator B wants to send a TLM-like communication to Simulator A. In this case, an asynchronous notify, on its own, is not sufficient, we also need some sort of asynchronous wait, to wait for that notification.
  - Finally (and similarly) of course, Simulator B could also arrive at the quantum boundary, and need to inform simulator A that they can now both continue to the next quantum (so long as there are no other simulators that are still busy). Again in this case an event notification would suffice, but requires not only the asynchronous notify, but also some means of asynchronously waiting for that event.

This paper proposes a mechanism that we would like to be standardized. In so doing, there are some extra requirements on our implementation:

- Above all, it must be consistent with what is already in the SystemC standard.
- It must support the widest possible different models and simulator integration schemes.
- While we propose an augmentation of the SystemC standard, the mechanism should work with foreign simulators as they are integrated with SystemC.

B. Implementation

The SystemC scheduler is a discrete event scheduler. The proposed asynchronous mechanisms has been (mainly) implemented inside the SystemC scheduler. The asynchronous mechanisms prevents SystemC from running out of events and stopping simulation.

New event classes have been added: `sc_async_event` and `sc_async_time` . They represent respectively an asynchronous event and an asynchronous time. The usage of asynchronous events are the same as normal events.

The way SystemC processes are declared as runnable hasn't been changed. However, new trigger type have been added in order to differentiate asynchronous events. The `wait` functions have been updated in order to support the new prototypes allowing waiting on asynchronous events or time.

A host lock mechanism has also been added to stop the SystemC thread in order to avoid the CPU spinning. The main change is in the simulation context. When there are no more SystemC events to run but asynchronous events are pending, the lock is set. The lock is released when an external event is posted in the queue through the asynchronous notify mechanism (already in SystemC 2.3.0). SystemC is then able to continue, evaluating runnable processes and continuing the simulation (or potentially locking again).

Before updating simulation time, or stopping simulation, a check with pending asynchronous events is done. Depending of their type (coupled with an event for example), the scheduler adapts the execution of events and can decide to lock simulation. The lock is only applied after the delta cycle runs, allowing eligible threads in the delta to run.

The current implementation has been evaluated with two SystemC kernels, and also with one SystemC kernel and six ISS using TLM quantums. The result shows, as expected, the SystemC thread was idle when it was waiting for external asynchronous events.

III. Quantum Time Based Events

The second issue this paper addresses is that of quantum time based events. TLM-2.0 introduced the concept of Quantums, allowing parts of a simulation to advance a local notion of time independently of one another. Doing so implies a number of notions of time within the same overall simulation.

Models that require synchronization with time (in order, for instance, to generate an event like an interrupt) have no alternative within the standard today than to use the SystemC kernel notion of time.

This means that models that use quantum time (as proposed by TLM-2.0) must frequently synchronize with the SystemC kernel time in order that time advances for these sorts of model. This is clearly sub-optimal, but also has wider reaching consequences in terms of the modeling style that is adopted.

However, typically, models of this sort are subservient to a system master, and effectively exist within a Quantum domain (a timer, for instance, operates within the quantum domain of a CPU master). Were the model to be able to find its Quantum domain, and register for timed events from that quantum domain, then the reliance on synchronization with the kernel would be removed.

A. Requirements

In order to achieve this, allowing models to register for Quantum time based events, models must be able to find their nearest local Quantum keeper. The Quantum Keeper must then provide a mechanism to allow for some notification mechanism to be triggered within the model.

Finally, a call to wait from within this mechanism should de-schedule the entire quantum domain from the SystemC kernel, (such that the quantum domain can be re-synchronized with the SystemC kernel from within a quantum time based event). This requirement is important as a model may discover, having been triggered from a quantum time based event that it needs to synchronize quantum time with the kernel time.

B. Implementation

This mechanism could be achieved using the full weight of SystemC methods and threads, adding the ability to register those methods and threads with an alternative time-wheel. However the purpose of this mechanism is not to build a totally parallel event wheel, but merely enable lightweight events to be triggered from the quantum time. Equally, arraigning the complexity of a full time wheel would add a significant body of code SystemC with little benefit, and arraigning that the requirement to be able to resynchronize between the quantum time and the SystemC kernel time would likely be more complicated. Not to mention that the resulting user API may well be confusing.

Therefore, our proposal is to allow lightweight callbacks to be registered with a Quantum keeper. This means that a wait called within a callback would cause the entire quantum domain to be de-scheduled as required.

The second consideration is how a model finds its local quantum keeper. Several proposals for this have been made. The quantum keeper, or even the callback as proposed by [3], can be passed along the TLM-2.0 transaction pathways. An alternative approach, which is more generic, but less flexible, would be to ensure that Quantum Keepers are, themselves, CCI parameters, which can be found through the normal CCI APIs. CCI is not addressed in this paper, but is a standard parameter interface that would allow models to search for quantum keepers by name, or position in the SystemC hierarchy.

We favor allowing both mechanisms. In retrospect, it is a pity that the time parameter in a `b_transport` call isnt actually a reference to the initiators quantum keeper. This would have been an elegant solution, however models have been written using the simple `sc_time` parameter. Our proposal is to add the reference to the initiators quantum keeper as an optional parameter to `b_transport` (it is not relevant for `nb_transport` models).

In addition, we propose that quantum keepers are CCI parameters, named with the string QuantumKeeper in their name.

C. Modeling style

One of the key concerns about synchronizing quantized models with the SystemC kernel has always been that the length of the quantum is non trivial to calculate. The paper [4] showed that a typical, simple, platform might have a bath-tub like performance as the quantum length increases. Finding the fastest performance in such a scenario is not straightforward. An alternative approach is not to use a fixed quantum at all, rather to require models to synchronize with the kernel only if and when that is required. This requires a fuller understanding of the system being designed (which typically is a very important thing for modeling engineers to understand!), and it is therefore actually valuable to insist that engineers do indeed add these synchronization points into their code. These synchronization points are typically required when there are interactions between different (parallel) systems. However, this approach, on its own, does not help when models need to be triggered from a local notion of time (e.g. the local quantum), such as a timer. In order to use this modeling style, its proponents have built their own mechanisms to enable local timed events. While this is perfectly reasonable, our aim is to ensure interoperability and standardization. Adopting a quantum based event mechanism will also allow the use of this modeling style while maintaining a common and standardized event API.

IV. Conclusion

SystemC has introduced a mechanism to notify a SystemC event in a thread safe manner. However, the SystemC kernel doesn't provide a mechanism to wait a SystemC event coming from another thread in a thread safe manner. In this paper, an exploration and an implementation of asynchronous features enabling an asynchronous wait in a standard manner has been presented. These mechanisms enable multiple simulators synchronization. The proposed generic mechanisms allow user to simulate multiple instances of SystemC kernels with different synchronization algorithms but also with external simulators (like ISS).

References

[1] G. Delbergue and M. Burton, "Inter-Simulation Synchronization - Asynchronous Wait." Presented at the 2016 SystemC Evolution Day, Munich, Germany, 2016.

[2] R. Dmer, "Seven Obstacles in the Way of Parallel SystemC Simulation." Presented at the 2016 SystemC Evolution Day, Munich, Germany, 2016.

[3] J. Engblom, "TLM Events - Making Temporal Decoupling Work." Presented at the 2016 SystemC Evolution Day, Munich, Germany, 2016.

[4] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego, "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01292317